Eliminating Embedded Software Defects Prior to Integration Test

Ted L. Bennett and Paul W. Wennberg
Triakis Corporation

Research has shown that finding software faults early in the development cycle not only improves software assurance, but also reduces software development expense and time. The root causes of the majority of embedded system software defects discovered during hardware integration test have been attributed to errors in understanding and implementing requirements. The independence that typically exists between the system and software development processes provides ample opportunity for the introduction of these types of faults. This article shows a viable method of verifying object software using the same tests created to verify the system design from which the software was developed. After passing the same tests used to verify the system design, it can be said that the software has correctly implemented all of the known and tested system requirements. This method enables the discovery of functional faults prior to the integration test phase of a project.

New, complex embedded systems are quick to take advantage of the unrelenting pace of advancement in computer hardware performance and capacity. Along with the increase in hardware capability comes a considerably greater increase in the functionality and complexity of the software in control.

Unfortunately, the methods and tools we use to develop and test systems and software have not kept up with the trend. This is evidenced by the number of software faults that pass undetected into the integration and operational phases of contemporary projects.

This is of concern for two important reasons. In the case of software in control of safety – or mission-critical systems – allowing a failure to pass undetected into the operational phase of a project may put lives and/or critical missions at risk. In all cases, the more faults that pass undetected into integration test and beyond, the more the project will cost and the longer it will take to complete.

This article presents a new, closed-loop method of simulating and verifying embedded system designs and their controlling software in a pure, virtual system integration laboratory environment. We have demonstrated and validated this method in a recently concluded research effort sponsored by the NASA Office of Safety and Mission Assurance under their Software Assurance Research Program [1]. Our investigation showed the following:

1. A new method of specifying, executing, and verifying an entire system design in a pure virtual environment.
2. How uninstrumented, embedded object software can be verified in the virtual system environment.
3. How the same tests used to verify the system design may be used to verify the controlling software.

   It follows from item No. 3 that if the software passes the same tests used to verify the system design then it correctly implements the known and tested system requirements. As a result, we now have a viable means of discovering requirements-induced software faults prior to the integration test phase of a project. This is significant because it has been shown that early discovery of faults reduces both project cost and duration.

Root Causes of Software Faults

The root causes of the majority of software defects discovered in integration test during the development of an embedded system have been attributed to errors in understanding and implementing requirements (see the sidebar "JPL Root-Cause Analysis of Spacecraft Software Defects" and Figure 1 on page 14). These may be the system and/or the software requirements. We assert that this is largely a result of the independence that exists between the requirements development and the software development processes.

The JPL report findings are echoed in reports of numerous other researchers such as Leveson [3, 4], Ellis [5], Thompson [6], and others. Consider some of the many avenues where requirements-related problems might be introduced:

- Assumptions/ambiguities affecting the interpretation of customer descriptions of desired system behavior.
- The difficulty in fully understanding the real-world environment in which the system will interact.
- The difficulty in anticipating all of the possible modes and states that the system may encounter.
- The difficulty in thoroughly validating and verifying requirements.
- Capturing accurate, unambiguous representations of requirements in a written document.
- Misinterpretation of system-level requirements by software designers.
- The difficulty in verifying that the design has correctly implemented the requirements.

To compound the problem, we generally cannot know at the onset of a project if we have accurately modeled the real-world system behavior. As a project advances, however, so does our understanding of the system. Additional faults may be introduced when subsequent refinements to the system model are not adequately communicated to the software development teams. To be more effective at creating software with a high level of assurance, not only must we reduce the number of errors attributable to misunderstanding and misimplementing requirements, but we must also improve communication between and among the system and implementation teams.

Shortcomings of Federated Development Methods

Contemporary, embedded systems development projects are typically conducted in a federated manner. In other words, the system and software development activities are conducted essentially independent of each other. To illustrate this point, Figure 2 on page 15 depicts the three principal loops comprising a typical project process. We will ignore hardware development activities since they are not germane to this discussion.

The first loop is where the system design is created. The system designers may make use of modeling, simulation, prototyping, executable specifications (ES), and other tools to satisfy the need to validate control algorithms, component interactions, etc. The system architects validate and verify their design through analysis, possibly tests, and possibly by similarity with reused components. They

JPL Root-Cause Analysis of Spacecraft Software Defects

In 1992, Dr. Robyn Lutz conducted an analysis for the Jet Propulsion Laboratory (JPL) to determine the root causes of the 387 software defects discovered during the integration test phase of the Voyager and Galileo spacecraft development efforts. The software controlling these spacecraft is distributed among several embedded computers with roughly 18,000 and 22,000 lines of source code, respectively. Lutz reported that the programming faults discovered on the two projects are distributed as shown in Figure 1.

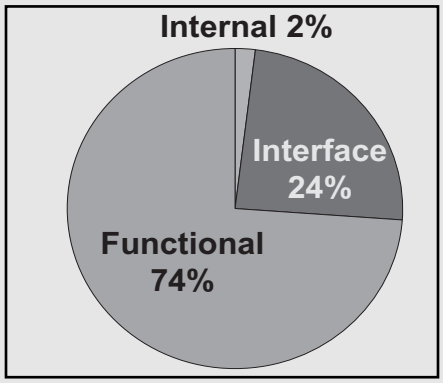


Figure 1: *Fault Distribution*

The fault classifications given in Figure 1 are defined as follows:

- *Functional faults* comprise the three subclasses listed below:
  a. Operating faults: Omission of, or unnecessary operations.
  b. Conditional faults: Incorrect condition or limit values.
  c. Behavioral faults: Incorrect behavior, not conforming to requirements.
- *Interface faults* are those related to interactions with other system components such as transfer of data or control.
- *Internal faults* are defined as coding faults internal to a software module.

The data show that 98 percent of the combined total software problems were classified as functional or interface faults that are directly attributable to errors in understanding and implementing requirements, and inadequate communication between development teams. Only 2 percent were due to software module coding errors [2]. The conclusions of the JPL report point to the need for improved focus in the following areas:

1. Interfaces between the software and the system domains.
2. Identification of safety-critical hazards early in the requirements analysis.
3. Use of formal (and unambiguous) specification techniques.
4. Promotion of informal communication among teams.
5. Keeping development and test teams apprised of changes to requirements.
6. Inclusion of requirements for *defensive design*.

then document the requirements for the implementation teams to follow. When satisfied with their design (or when time runs out), the system team delivers the system specification package to the implementation teams.

Entering the second loop shown in Figure 2, the software implementation team interprets the relevant requirements – whether written in natural language, specification design language, or executable specifications – derives software requirements, and creates its design. The software developers write their own tests to verify conformance to the requirements as they have interpreted them. They may use some form of simulation, hardware development boards, inspection, analysis, or similarity comparison to facilitate verification of their code.

When a major part of the system functionality has been coded, the software team creates a build. The software is loaded into its target hardware where integration test begins in the laboratory. Connected to test equipment, simulators, and perhaps other system elements, the control software is stimulated by the hardware environment under the control of custom test software. Bugs discovered during integration test are filed as problem reports and passed back to the development team to resolve, thereby completing the third loop.

We see the independence that exists between the system and software loops in this development process as the primary reason for the propagation of software faults into integration test. Further, this independent process may breed duplicity of effort where the software and system teams write their own tests to verify the same behavior at the system and software levels.

Our research has shown a method of connecting the system and software development loops that allows tests written for system verification to be used to verify the software itself. This enables the software to be thoroughly debugged in a pure, virtual environment before it ever gets to the hardware integration phase.

Coupling the System and Software Development Loops

Figure 3 illustrates our approach to connecting the system and software development loops. This new approach retains the system and software development loops, but eliminates the loop where the hardware integration lab is used for software debug activities.

As before, your project begins with the development of a system design using various tools for algorithm development, etc. However, in lieu of passing the design and requirements to the implementation teams as a collection of disparate specifications, the entire system and the environment in which it interacts is simulated using a form of ES. All parts in the simulation are bounded like their real-world counterparts so the interface behavior of each element can be correctly modeled and specified. Parts are created with built-in failure modes that may be activated under test control.

Having modeled the behavior of the entire system environment, you now have a complete virtual system integration laboratory (VSIL) in which to validate and verify your system design. The next step is to create a suite of tests based upon nominal and off-nominal scenarios for which the system has been designed to react. Our testing philosophy is to exercise the system by driving the environment as realistically as possible, and monitoring the system behavior in response. This is generally not a viable approach for hardware system integration laboratory setups due to the cost or difficulty involved in procuring, creating, and synchronously controlling all the disparate pieces of hardware and simulators necessary to realistically drive the target system.

The completed and verified VSIL is then passed, along with the system-level tests and any supplemental written requirements, to the development teams. The teams create hardware and software designs from the specified processing, communication, interface, control, and other requirements. As soon as the hardware architecture has been established, the target embedded controller for which the software is being developed must be simulated with sufficient fidelity to run the unmodified object software. Because the simulated controller hardware is bounded (i.e., it has identical interfaces) like the ES part from which it was developed, it may be plugged into the VSIL in

place of its ES counterpart. We refer to this controller hardware simulation part as a detailed executable (DE) (see Figure 3).

The DE gives the software team the ability to test the software it develops (see Figure 3, step 1) in the VSIL (see Figure 3, steps 2-4). After replacing the controller ES with the DE, the software being developed may be compiled and loaded into the DE at any time for testing in the VSIL. All of the tests created to verify the system design can be used, without modification, for software verification. Additional tests must be added to verify that software has correctly implemented lower-level requirements whose detail has not been addressed at the system level (e.g., built-in test, etc.).

After running the desired tests, the software development team analyzes the results and determines the cause of any failures. The team then corrects any identified faults, recompiles the revised modules, and retests the build in the VSIL (see Figure 3, steps 1-4). In practice, step 3 is performed once since the DE becomes an integral part of the VSIL following replacement of its ES counterpart. The VSIL is tightly coupled with the integrated software development environment used by the software team – thereby facilitating the code/compile/load/verify process.

Some of the problems discovered may require the attention of the system designers. When this necessitates a system design change, the VSIL is revised and tested and redistributed to the software development teams. In this manner, the software is always developed and tested in the most current system design – thereby eliminating the possibility of software problems being introduced due to miscommunication of system design changes.

The software design/code/verify/debug loop is repeatedly executed until the final build passes all tests and until all paths through the code have been exercised in the VSIL. Thus, the software has been thoroughly verified and is ready for integration testing with the real flight hardware.

It is worth noting that since the object code itself is tested in the VSIL, the real-time operating system (RTOS), any reused/commercial off-the-shelf modules, and all newly developed software are verified together in the virtual target environment. The VSIL itself is a Microsoft Windows-compatible application that interfaces with standard integrated development environment tools. A VSIL is as easily used as a typical lab test setup (e.g., emulator, simulators, target hardware) and readily distributed to all project development personnel. Since the entire system and environment are modeled in the VSIL, modifications and refinements can be coded, validated, verified, and distributed to the entire team. VSIL revisions and verification tests may be controlled using standard configuration management tools and techniques. Lastly, the VSIL is purely virtual: no hardware is required other than the Windows-based PC on which it runs.

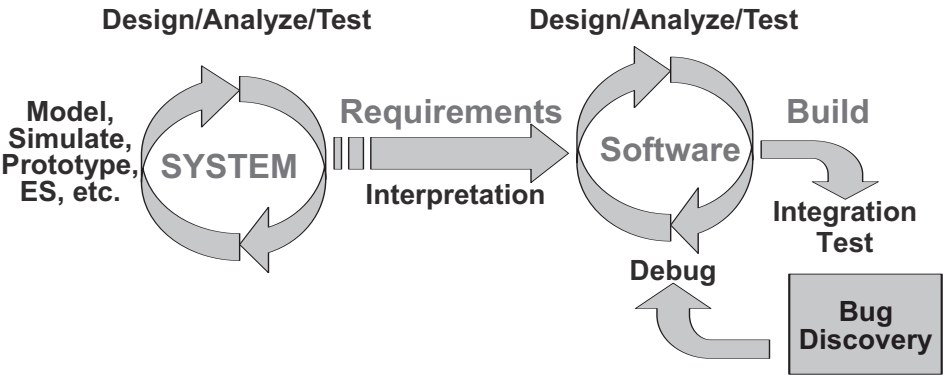


Figure 2: *Federated Development Process*

Discussion

We have presented a new method of embedded systems and software verification and validation (V&V) that closes the loop between system and software development activities. In so doing, the system and software development processes can now be connected through common verification tests.

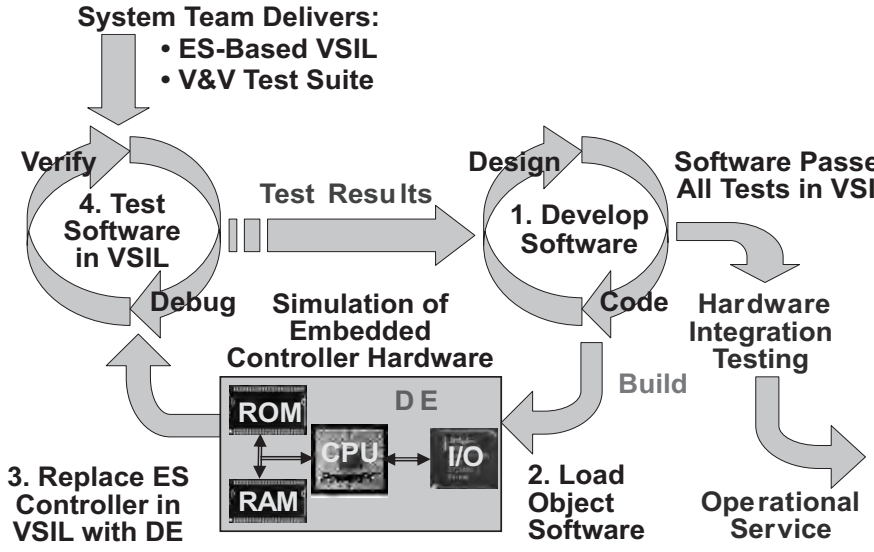
Finding and repairing software faults early in the project development cycle can lead to substantial savings (see the sidebar "Economics of Fault Finding" on page 16). For example, requirements and communication-induced errors like 98 percent of those discovered during the integration phase of the Voyager and Galileo spacecraft software projects, can be found and repaired at one or perhaps more orders of magnitude lower cost.

Implications

Below is a summary list of some of the ways that the methods presented in this article may be of economic benefit to embedded software development:

1. Discover system errors early in the development cycle where it is least costly to correct them.
2. Reduce interpretation-induced software faults due to ambiguities in system requirements.
3. Improve ability for dynamic, non-invasive test of system and software response to failure conditions.
4. Reduce software faults caused by breakdown in communication of system requirements changes.
5. Utilize new capacity for empirical software V&V in cases where analysis was the only viable means, for example, realistic fault injection and failure mode testing, complex digital signal processor designs, etc.
6. Provide a highly viable means of verifying automatically generated code,

Figure 3: *Closed-Loop Software Verification in Virtual System Integration Lab*



Economics of Fault Finding

Estimates of the cost to find and correct software faults at each of the principal stages of a project have been publicized and widely referenced since 1976 when Boehm first published his study [7] on the subject. Cost numbers vary depending on the type of application for which the software is being developed, but the common thread they all exhibit is the substantial increase in project costs caused by carrying problems from one development stage to the next.

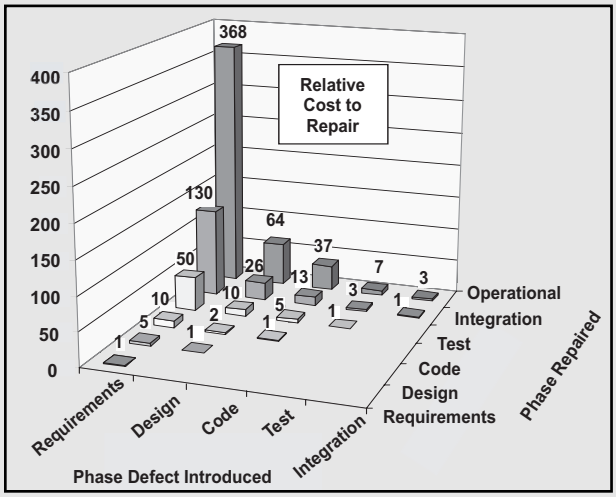
A report released in May 2002 by the National Institute of Standards and Technology (NIST) [8] contains a thorough analysis concluding that inadequate software testing costs the United States an estimated *$59.5 billion annually*. The 309-page NIST report is a well-considered treatise on the economic impact of inadequate software testing.

While these numbers are extrapolated from software developed for the financial services and transportation applications (computer-aided design, computer-aided manufacturing, etc.) sectors, the message applies even more significantly to industries engaged in developing software for safety and mission-critical applications such as aerospace, medical, defense, automotive, etc. Failures of safety/mission-critical software may result in harm to, or loss of human life and/or mission objectives such as in the case of the Therac-25 radiation overdose accidents [2] and the Ariane-5 maiden launch failure [9]. The Therac-25 software caused severe radiation burns in numerous cancer patients before it was implicated. The cost of allowing the Ariane-5 software defect to pass into the operational phase has been estimated to be as high as $5 billion alone.

NASA recently sponsored a study to evaluate the economic benefit of conducting independent validation and verification during the development of safety-critical embedded systems [10]. This study presented cost-to-repair figures focused specifically on embedded systems projects. Figure 4 shows the relative cost to repair factors – considered to be conservative estimates for embedded systems – used in this study.

The graph in Figure 4 tells us that an error introduced in the requirements phase will cost five times more to correct in the design phase than in the phase in which it was introduced. Correspondingly, it will cost 10 times more to repair in the code phase, 50 times more in the test phase, 130 times more in the integration phase, and 368 times more when repaired during the operational phase. The graph also gives the cost multipliers for problems introduced in the design, code, test, and integration phases of the development cycle.

Figure 4: *Relative Cost of Software Fault Propagation*



reused software, and RTOS.

Creating a system design with the type of ES discussed herein results in a verifiable system architecture that is readily translated into component- and interface-level designs. When contracting out the development of subsystem software, the system-level verification tests can provide an excellent way to assure that the contractor has developed the software correctly.

Because ES parts may be created with intrinsic failure modes that can be invoked dynamically under test control, the system designer can empirically verify the specified system response to a variety of off-nominal conditions. This ability allows greater latitude in the type and number of tests that can be conducted when compared with what is economically viable in a hardware integration lab.

Verifying the VSIL

The VSIL is, in fact, a model of both the system being developed and the environment in which it is designed to interact. Before it can be of use, we must have confidence that the VSIL represents its target adequately.

We have adopted an effective approach that is perhaps best described as test as you go. As parts are simulated to implement specific requirements, system-level tests are created simultaneously to verify that they behave correctly. Part functionality may be developed and tested incrementally as requirements are implemented. At the end of this process, all VSIL parts have been implemented and verified and a basic set of system-level tests has been developed.

Parts developed to a high fidelity level may require a supplemental verification activity where the real-world equivalent part is used for comparison purposes. In the case of developing an instruction-set-level CPU simulation, we run test code designed to verify instruction execution on a hardware development board and compare the results with the outcome of running the same code on the simulated part. The CPU parts we have developed are not cycle-accurate but are refined to where the instructions execute within an average of 4 percent of the hardware performance (works well for embedded software verification). This is in keeping with our philosophy of not implementing greater fidelity than necessary.

VSIL Development Tool

Triakis developed its first avionics simulator more than a decade ago to save time verifying software modifications and to avoid contention for lab test resources. This initiative spawned the creation of IcoSim, Triakis' general-purpose simulator development tool, and its companion software developer's kit (SDK).

The IcoSim SDK is typically available at no cost to customers availing themselves of Triakis' VSIL development services. In the second quarter of 2006, however, Triakis plans to make IcoSim freely available to the general public by turning IcoSim into an open source project[1] whose use will be governed under a Lesser General Public License (LGPL) [11]; simulated parts will be covered individually under a LGPL, GPL [12], or proprietary license.

Tool Description

Since it is destined to become an open source project, the descriptive details provided herein are intended to promote an understanding of how we accomplish what we have presented.

Written in C++ and C, IcoSim allows the use of diverse part types ranging from low to high abstraction levels. It also supports using mixed mode parts such as analog, digital, mechanical, hydraulic, magnetic, electromagnetic, et al.

IcoSim is well suited to creating a VSIL for use in developing embedded

systems and software because the simulated parts may be bounded exactly like their real-world counterparts. In other words, the inputs and outputs of each virtual part are readily modeled after the behavior of their real-world part's digital, analog, mechanical, etc. input/output. Once its behavior is verified, a virtual part may be identified with the same part number as its counterpart, and repeatedly used wherever system designs specify.

VSIL Parts Libraries

In addition to the NASA research that validated the methodology presented, IcoSim has been used to create VSILs for software verification on more than two dozen avionics projects over the past decade. It is scalable to any size system and has been used for verification of software in single and dual-redundant avionics systems ranging in criticality from Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B[2], level A (safety-critical) to level D (low criticality). It has also been used for verification of embedded digital signal processor (DSP) software implementing Kalman filter algorithms.

Triakis' parts library includes instruction-set-level simulations of many microprocessors in use today such as the MPC555, MPC750, RAD6000, MC68000, MC68332, DSP56005, DSP56302, DSP56309, I80196, I8051, I8096, I8097, I8798, et al. Numerous additional peripheral and glue parts are in the library as well as a host of actuators and sensors that have been created in support of various VSIL projects. Triakis has also created a collection of parts that simulate many different data buses and protocols, e.g., Aeronautical Radio, Inc. (ARINC) 419; ARINC 739; Military-Standard-1553; Time-Triggered Protocol; Avionics Standard Communication Bus; Commercial Standard Data Bus; Avionics Full-Duplex Switched Ethernet; Serial Peripheral Interface; Peripheral Component Interconnect, Controller Area Network, etc.

To support testing with a VSIL, we have simulated standard laboratory test equipment such as oscilloscopes, signal generators, and the functional capability of microprocessor emulators. The VSIL is an ideal environment for gathering dynamic software metrics without instrumenting either the target operating system or the software. Code path coverage, Modified Code Decision Coverage reports, throughput analysis, timing analysis, and many other helpful reports are readily produced in this environment with the addition of instructions to the test script.

Costs of VSIL Development

A VSIL is made by interconnecting objects at the lowest level of abstraction to make successively higher levels of functional parts until the required environment is complete. This hierarchical, modular approach maximizes the potential for part reuse on subsequent development projects.

To be efficient at making a VSIL, each part is simulated only to the level of fidelity necessary to achieve one's goals. For example, an aircraft rudder is attached to a sensor that reports its angular position to avionics subsystems as required. The sensor has a mechanical link to the rudder, has inertial properties, may have inductive coils, may have an armature, may be excited by a 400 Hz reference, etc.

"There are many factors that influence the cost, but a typical VSIL [virtual system integration laboratory] can be developed for about 5 percent to 10 percent of the overall project cost."

While we could model all of these characteristics with great precision, it would be a waste of effort if our system only required the correct transfer function of rudder angle to sensor output at a given update rate. Since part fidelity is directly proportional to effort, being selective about where to incorporate higher fidelity is key to cost-effective VSIL creation.

It is difficult to quantify the costs of creating a VSIL for system and software development because of the large number of variables involved such as the following:

- System size.
- System complexity.
- Number of parts to be simulated.
- Number of control processing units to be simulated.
- Experience of simulation engineer(s).

Because of the part-oriented nature of the VSIL, the cost of creating a simulator for a given project will vary in proportion to the number and complexity of new parts that must be created. Many new, embedded designs reuse proven design elements from prior projects so the cost of developing simulators diminishes with successive applications.

Supplemental VSIL Benefits

The benefit of using a VSIL for embedded systems and software development increases with project size, system complexity, and geographic diversity of organizations and personnel contributing to the project.

In addition to the cost benefits of early software fault discovery, a VSIL can support a project in other important ways. Some of these benefits are directly measurable, but others may have less tangible value:

- When contracting out development of a subsystem, supplying the vendor with a VSIL and its system test suite can be a highly effective means of verifying that the software conforms to the requirements.
- Development teams in local and remote locations can quickly re-verify their software following system revisions that have been implemented and tested in a VSIL. Using standard configuration control procedures, the latest system revision can be distributed to all teams as soon as it is available.
- Providing a VSIL to every programmer promotes a broader, big-picture understanding of the system. Every programmer tests on the whole system, every time.
- Testing in a VSIL reduces the dependence on laboratory test stations; consequently, fewer are required.
- Less dependence on laboratory test equipment reduces resource-contention delays during development.
- A VSIL may be helpful in the operational phase of a project for the following:
  - Software re-verification following upgrade modifications with full regression testing.
  - Re-verifying software on obsolescent-driven hardware design changes.
  - Verification of system compatibility with upgrades to peripheral or subsystem units.
  - Eliminating or reducing reliance on test equipment setups that must be maintained to support software changes following entry into service.

While not a rigorous analysis, one avionics company's post-project review of having used a VSIL for verification of their dual-redundant avionics software revealed some attractive cost-benefits.

Based on their findings they concluded that future projects could expect a 24 percent schedule savings, a $130,000 direct savings on laboratory equipment, and realize an overall cost savings of 14 percent on an average $4.5-million project. These estimates do not take into account the benefits afforded by a VSIL throughout the operational life of a product. There are many factors that influence the cost, but a typical VSIL can be developed for about 5 percent to 10 percent of the overall project cost. This places the return on investment in the range of 40 percent to 180 percent for the above project.

Experiences will no doubt vary from project to project; however, these estimates can provide useful guidance when assessing the life-cycle cost/benefit of using a VSIL for development.

Summary

The new method of embedded systems and software V&V presented here goes far beyond an incremental improvement to the status quo. While not a panacea, it does provide a cost-effective, proven means of the following:

- Ensuring that the target software has implemented all known and tested system requirements – prior to hardware integration.
- Verifying automatically generated code, reused software, and the RTOS.
- Verifying response of systems and software to a wide range of realistic, dynamic failures and off-nominal scenarios.
- Re-verifying software following system revisions and updates.
- Ensuring that hardware redesigned for obsolescence is compatible with the software.
- Verifying that new and upgraded peripherals and subsystems function correctly with the target system.

The approach described provides a bridge between algorithm and model development tools, and the real-world system environment in which embedded algorithms must function. This method is a highly viable way to address a number of problems that hamper efficient embedded systems and software development.◆

References

1. Bennett, T.L., P.W. Wennberg. "The Use of a Virtual System Simulator and Executable Specifications to Enhance Software Validation, Verification, and Safety Assurance – Final Report." Software Assurance Research Program Results Web Site. Fairmont, West Virginia : NASA IV&V Facility, June 2004. <http://sarpresults.ivv.nasa.gov/ViewResearch/285/32.jsp>.
2. Lutz, R.R. "Analyzing Software Errors in Safety-Critical, Embedded Systems." Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, 1994.
3. Leveson, N.G. Safeware – System, Safety, and Computers. Addison-Wesley, 1995.
4. Leveson, N.G. "Software Safety: What, Why, and How." ACM Computing Surveys 18.2 (1986).
5. Ellis, A. Achieving Safety in Complex Control Systems. Proc. of the Safety-Critical Systems Symposium. Brighton, England: Springer-Verlag, 1995: 2-14.
6. Thompson, J.M. "A Framework for Static Analysis and Simulation of System-Level Inter-Component Communication." Masters Thesis. University of Minnesota, 1999.
7. Boehm, B.W. "Software Engineering." IEEE Transactions on Computer 1.4 (1976): 1226-1241.
8. Tassey, G. "The Economic Impacts of Inadequate Infrastructure for Software Testing." National Institute of Standards and Technology, 2002 <www.nist.gov/director/progofc/report02-3.pdf>.
9. Leveson, N.G. "The Role of Software in Spacecraft Accidents." AIAA Journal of Spacecraft and Rockets 41.4 (July 2004).
10. Dabney, J.B. "Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report." Fairmont, W.V.: NASA IV&V Facility, 2003. <http://sarpresults.ivv.nasa.gov/ViewResearch/289/24.jsp>.
11. GNU Lesser General Public License. Vers. 2.1. Boston, MA: Free Software Foundation, Inc., 1999 <www.opensource.org/licenses/lgpl-license.php>.
12. Open Source. The General Public License (GPL). Vers. 2. Boston, MA: Free Software Foundation, Inc., 1991 <www.opensource.org/licenses/lgpl-license.php>.

Notes

1. Details about open-source projects can be found at <http://sourceforge.net/>.
2. Information about DO-178B can be found at <www.software.org/quagmire/descriptions/-178b.asp>.

About the Authors

Ted L. Bennett is director of Systems Engineering and Business Development at Triakis Corporation. He has more than 25 years experience in embedded hardware and software design, systems engineering, project management, and business development in the aerospace industry. Bennett was principal investigator for the NASA-sponsored research project that validated the breakthrough methodology presented in this article. He is also principal investigator on two additional NASA research grants currently being conducted by Triakis. He has a Bachelor of Science in electrical engineering from the University of Wisconsin at Madison.

Triakis Corporation
16149 Redmond WY STE 177
Redmond, WA 98052
Phone: (425) 558-4241
Fax: (425) 558-7650
E-mail: ted.bennett@triakis.com

Paul W. Wennberg is president and founder of Triakis Corporation and conceived and created IcoSim, the pure virtual environment simulator tool discussed in this article. He has over 20 years experience in the design and test of embedded systems hardware and software, and pioneered this new methodology. A U.S. Air Force veteran, Wennberg logged over 1,400 hours piloting T38 and KC135 aircraft prior to completing his service with the rank of captain. He has a Bachelor of Science in electrical engineering from the University of Washington at Seattle.

Triakis Corporation
16149 Redmond WY STE 177
Redmond, WA 98052
Phone: (425) 861-3860
Fax: (425) 558-7650
E-mail: paul.wennberg@triakis.com